

# **ManPy Documentation**

## Table of Contents

1	Introduction and Scope .....	3
2	How to get started.....	4
3	Architecture.....	5
3.1	ManPy Generic (Abstract Classes) .....	5
3.1.1	CoreObject.....	6
3.1.2	ObjectInterruption .....	9
3.1.3	Entity .....	10
3.1.4	ObjectResource .....	10
3.1.5	Auxiliary .....	10
3.2	Expanding the Code .....	11
4	Examples .....	13
4.1	A single server model .....	13
4.2	Two servers model with failures and repairman.....	14
4.3	An assembly line.....	17
4.4	Parallel stations and Queue customization .....	18
4.5	Parallel stations and counting the parts of each machine .....	22
4.6	Stochastic model .....	24
5	List of Objects .....	<b>Error! Bookmark not defined.</b>

# 1 Introduction and Scope

ManPy stands for "Manufacturing in Python" and it is a layer of Discrete Event Simulation (DES) objects built in SimPy (<http://simpy.sourceforge.net/>). The current version of ManPy is based on SimPy2 (<http://simpy.sourceforge.net/old/>). This happens because at the time ManPy implementation progressed, the newest version was not available. We plan to progress to SimPy3 (<http://simpy.readthedocs.org/en/latest/>) soon. This is not supposed to affect in a great extend this documentation.

The scope of the project is to provide simulation modellers with a collection of open-source DES objects that can be connected like "black boxes" in order to form a model. This collection is desired to be expandable by giving means to developers for:

- customizing existing objects by overriding certain methods
- adding brand new objects to the list

ManPy is product of a research project funded from the European Union Seventh Framework Programme (FP7-2012-NMP-ICT-FoF) under grant agreement n° 314364. The project name is DREAM and stands for "*Simulation based application Decision support in Real-time for Efficient Agile Manufacturing*". More information about the scope of DREAM can be found at <http://dream-simulation.eu/>.

DREAM is a project which kicked off in October of 2012 and finishes in September of 2015. ManPy is an ongoing project and we do not claim that it is complete or bug-free. The platform will be expanded and validated through the industrial pilot cases of DREAM. Nevertheless, it is in a quite mature state to attract the interest of simulation modellers and software developers.

The dream repository contains the following 3 folders:

- **platform**: contains code for a GUI that is being build for ManPy. This is a parallel work and it is not always synchronized to ManPy's latest version
- **simulation**: contains all the simulation ManPy code along with some input files and some files from a commercial simulation package that are used for verification
- **test**: contains unit-tests for the project.

This document regards ONLY the ManPy part of the project. Note that ManPy is independent from the GUI and can be used separately as a library of simulation objects, which can be used to form a model. Users can implement alternative methods to be able to construct models, run them and get results.

The reader of this documentation needs to have basic, yet not deep knowledge of programming in Python (<http://www.python.org/>) and SimPy2. Also the reader is expected to have a basic knowledge of the Discrete Event Simulation (DES) technique.

## 2 How to get started

To be able to run the documentation examples just copy the dream/simulation to your Python folder. Then you can import ManPy objects as it is written in the examples, e.g.:

- *from simulation.Queue import Queue* or
- *from simulation.imports import Machine, Source, Exit*

If you copy the whole dream folder, then the import should change to something like *from dream.simulation.Queue import Queue*. If you want to rename the folder it can be *from path.to.my.new.folder.Queue import Queue*.

ManPy uses the following Python libraries:

- SimPy2
- NumPy
- SciPy
- xlrd
- xlwt

### 3 Architecture

ManPy objects are written exclusively in Python and they use methods of SimPy. Figure 1 shows the current state of the architecture.

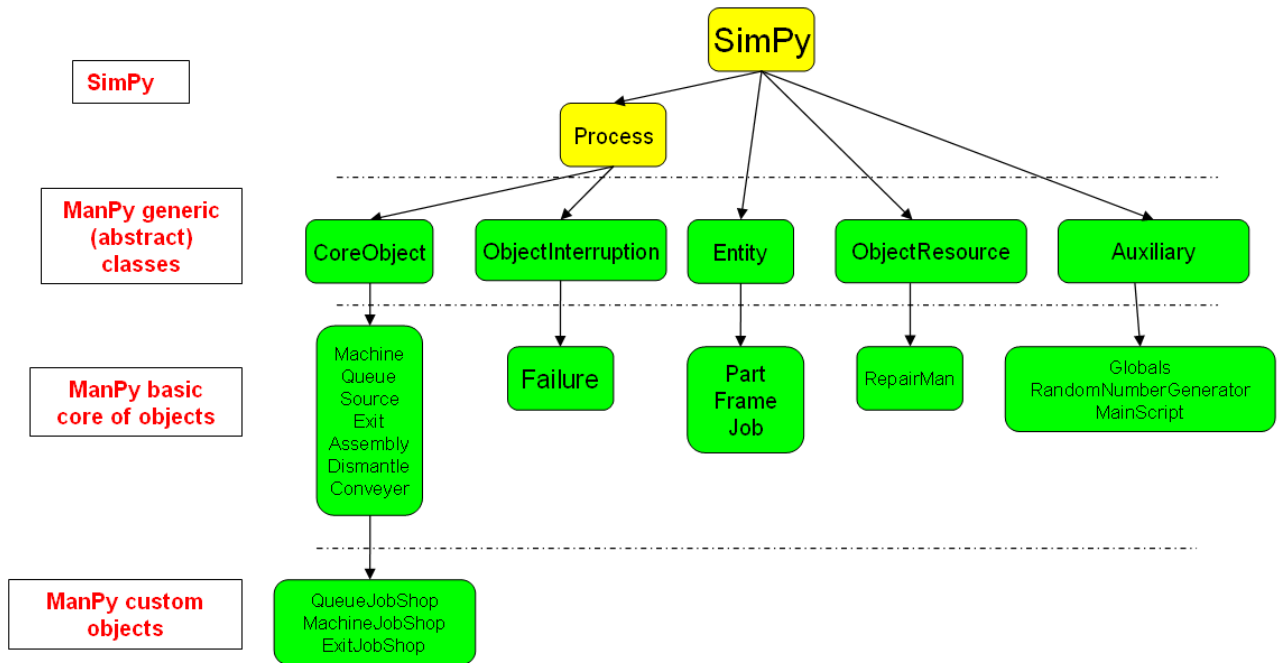


Figure 1: The ManPy class hierarchy.

In Figure 1 four different layers are depicted:

- On the top we have SimPy classes
- The top layer of ManPy is a set of generic, abstract classes. There are not supposed to have instances, nevertheless they are important because:
  - they help in the grouping of objects
  - generic methods are defined for all those classes which the simulation objects inherit and override
- Below the generic objects lies the basic core of ManPy objects. This is currently being populated and expanded.
- On the bottom we have custom objects of ManPy. These inherit from one object of the basic core and customize it according to the needs of the modeller

In the remaining of this chapter the generic classes of ManPy will be described.

#### 3.1 ManPy Generic (Abstract Classes)

The layer of abstract classes is the “heart” of ManPy. These give the basic guidelines of how the platform is structured. Note that since this is an ongoing work, the names of the classes may change, since we currently think towards the best abstraction. Also new generic classes might be added in future versions, even though the number should be kept reasonably short. The abstract classes include:

- **CoreObject**: all the stations in a model that are permanent for the model. These can be servers or buffers of any type.
- **ObjectInterruption**: all the objects that can affect the availability of another object. For example failures, scheduled breaks, shifts etc.
- **Entity**: all objects that get processed by or wait in CoreObjects and they are not permanent in a model. For example parts in a production line, customers in a shop, calls in a call centre etc.
- **ObjectResource**: all the resources that might be necessary for certain operation of a CoreObject. For example repairman, operator, electric power etc. An ObjectResource is necessary in modelling when two or more CoreObjects compete for the same resource (e.g. two machines competing for the same operator).
- **Auxiliary**: These are auxiliary classes that are needed for different simulation functionalities. Unlike the other categories described here, auxiliary classes do not inherit from one parent class, even though it is depicted in such a way in Figure1 for reasons of coherence.

In the following subsection each category of generic classes will be described in more depth.

### 3.1.1 CoreObject

As CoreObjects are categorized all the stations in a model that are permanent for the model. These can be servers or buffers of any type. It is in the philosophy of ManPy that the CoreObjects will handle most of the simulation logic, so that a more generic process oriented approach is achieved. CoreObjects inherit from SimPy.Process class.

CoreObjects should be able to communicate no matter what their type is. For example, a Machine should be able to retrieve an Entity from another CoreObject, using the same code, no matter if this CoreObject is a Queue or also a Machine. For this reason all the CoreObjects implement a set of methods which have the same name, but different implementation for every object. This set of methods includes:

- Definition methods**: are used for the instantiation of the object
- Transaction methods**: are used to define how the objects exchange entities
- Control methods**: are used to retrieve the state of an object
- Supplementary methods**: are used to define certain objects in Transactions or Control methods
- Output and calculation methods**: are used either to output results or trace in different formats or to make certain calculations
- Main simulation method**: just one method that is used to control the progress of the object in the simulated time. In Python terms this is a generator method and it is the only one where the *yield* commands of SimPy can be invoked

Also, CoreObjects share some conventions for certain variables listed below:

- *Res*: this is an instance SimPy.resource type. It keeps the Entities that the CoreObject holds in its activeQ, in which we also refer to as the “internal queue” of the core object in this documentation (not to be confused with the Queue object).
- *next*: a list that holds all the successors of the CoreObject, i.e. the CoreObjects to which the object can give an Entity.
- *previous*: a list that holds all the predecessors of the CoreObject, i.e. the CoreObjects from which the object can receive an Entity.
- *successorIndex*: an integer that points to the successor of *next* that the CoreObject can give an Entity at every moment off simulation time.
- *predecessorIndex*: an integer that points to the predecessor of *previous* from which the CoreObject can receive an Entity at every moment off simulation time

Note that *next* and *previous* lists may be empty. This can happen for several reasons:

- For certain objects it is not logical to have both lists. For example an Exit object should not have any successors
- Sometimes the flow is completely dependent on Entities attributes (e.g. in a jobshop). In such cases objects do not need to have predecessors or successors.

CoreObjects share also several other attributes that hold certain important values. For example *timeLastEntityEntered* holds the simulation time that an Entity entered in the CoreObject. Also they have counters that hold certain results. For example *totalFailureTime* holds the failure time for a CoreObject, which can be divided by the length of the simulation run in order to give the percentage of time that the CoreObject was in failure. The list of such variables is currently populated.

Below we will discuss the methods of each of the 6 categories of generic methods.

#### 3.1.1.1 Definition methods

These are used for the instantiation of the object. 3 such methods exist:

- *\_\_init\_\_*: this is the python constructor method. This method is ran when the instance is created.
- *initialize*: this method initializes the object for a simulation replication. It should not be confused with the constructor above. The constructor is ran only in the creation of the object, while initialize must be ran in the beginning of every replication.
- *defineRouting*: it defines the *next* and *previous* lists, i.e. successor and predecessor objects.

#### 3.1.1.2 Transaction methods

These handle the transactions of Entities between CoreObjects. In every transaction two CoreObjects take part. The giver object is the one that gives the entity and the receiver object is the one that obtains it.

Two such methods exist:

- *removeEntity*: it is ran on the giver object and it removes an Entity from it. The objects sort the Entities they hold in such a way, so that the object that will be removed is the first object of the internal queue.
- *getEntity*: it is ran on the receiver object and it obtains an Entity from the giver. In essence it calls the *removeEntity* method of the giver object and adds the Entity to each internal queue

#### 3.1.1.3 Control methods

For every object they provide information about its state. They return true or false. 3 such methods exist:

- *canAccept*: returns true if the object is in a state to receive an Entity. The logic depends on the type of the object. For example in a Queue the capacity might need to be checked, while an Exit object might always be in the state of receiving an Entity. Note that sometimes it is needed that this method should return true only to the object that it can receive the Entity from. In this case, the object that calls the method must be passed as an argument.
- *haveToDispose*: returns true if the object is in a state to give an Entity. The logic depends on the type of the object. For example a Queue may need to check only if it does hold one or more Entities, while a Machine might need to check also if the Entity that it holds has ended its processing. Note that sometimes it is needed that this method should return true only to the object that it can give the Entity to. In this case, the object that calls the method must be passed as an argument.
- *canAcceptAndIsRequested*: returns true only when both conditions are satisfied: the object is in the state to accept an Entity and also another object is requesting to give one Entity to it. As we will see, only when this method returns true the main simulation logic of the object is started.

Note that contrary to the other methods described in this section, this one is expected to be called only internally from an object (there is no need for a *CoreObject* to call *canAcceptAndIsRequested* of another). So it is not obligatory that the name is the same for all *CoreObjects*. Nevertheless, for reasons of coherence this method is mentioned here and we believe that all the *CoreObjects* should keep the naming convention.

#### 3.1.1.4 Supplementary methods

These methods are used to obtain specific objects that are needed for the transaction and control methods. Six such methods exist:

- *getActiveObject*: returns the active object in the transaction. This always returns *self*, and they can be used interchangeably (though *self* should be faster since it does not call a method).
- *getActiveObjectQueue*: returns the internal queue of the active object. This always returns *self.Res.activeQ*, but it is preferred to use the method since it makes the code cleaner and lesser need of knowledge of the internals of *ManPy* is achieved.
- *getGiverObject*: returns the giver object in a transaction.
- *getGiverObjectQueue*: returns the internal queue of the giver object in a transaction.
- *getReceiverObject*: returns the receiver object in a transaction.
- *getReceiverObjectQueue*: returns the internal queue of the receiver object in a transaction.

#### 3.1.1.5 Output and calculation methods

Perform calculations or output data. Five such methods exist:

- *sortEntities*: it sorts the Entities in the internal queue of the *CoreObject*. Many times this method might not be needed. However, there are times when it is essential. E.g. when a Queue needs to sort its Entities according to a predefined rule.
- *calculateProcessingTime*: Calculates the processing time every time one Entity gets into the *CoreObject* for processing.
- *postProcessing*: is called for every object in the end of a simulation replication, The purpose is to perform certain calculations. For example, if a Machine is still processing an Entity when the simulation ends, this processing time should be added so that the results are accurate. Note that when an object is complex, sometimes it is difficult to debug such a method. On the other hand, in a long simulation run a mistake in this method would most probably not introduce a large error.
- *outPutResultsJSON*: outputs the results of the object in a JSON format. All the objects output to the same JSON file. If we have more than one replications, the results are given in confidence intervals.
- *outPutResultsXL*: outputs the results of the object in an Excel file. All the objects output to the same Excel file. If we have more than one replications, the results are given in confidence intervals. To save the excel file the user should add *G.outputFile.save("filename.xls")* in the main script.
- *outPutTrace*: outputs trace in an Excel sheet when an important event happens (e.g. an Entity gets into the *CoreObject*). All the objects output to the same Excel file and the events are sorted in increasing timestamp. The trace is essential for debugging. To run a model that is believed to be verified, it should be turned off since it slows the program significantly.

Note that contrary to the other methods described in this section, the 3 output methods and *postProcessing* are expected to be called only internally from an object or from a main script (e.g. there is no need for a *CoreObject* to call *outPutResultsJSON* of another). So it is not obligatory that the name is the same for all *CoreObjects*. Nevertheless, for reasons of coherence these methods are mentioned here.



ManPy users are invited to write new methods for objects, in case they desire to output results in different format (e.g. XML). Also it is logical that ManPy users would like to override these methods to customize the results that they get.

#### 3.1.1.6 Main simulation method

Here the logic that the `CoreObject` follows as it evolves through time. There is only one such method:

- *run*: this is a generator method and it is the only one where the *yield* commands of SimPy can be used. For this reason *run* requires that the user knows the internals of SimPy in order to customize. It is common (but not obligatory) that in such a method there is a *while* loop that runs all through the simulated time. The logic followed in every `CoreObject`'s *run* method is:
  1. Wait until *canAcceptAndIsRequested* becomes true.
  2. Call object's *getEntity* method so that it obtains the Entity from a giver object.
  3. Carry on the logic of the object (unique for every different type).
  4. When the process is ended the *haveToDispose* method should return true to the object that can be a receiver. Of course this can change again (e.g. from a failure while the machine was blocked).
  5. Stay there. The loop cannot start again if it should not, since step 1 takes care of it. When at some point some receiver object calls the `CoreObject`'s *removeEntity* method this may change and the loop will restart.

#### 3.1.2 ObjectInterruption

As `ObjectInterruptions` are categorized all the objects that can affect the availability of another object. For example failures, scheduled breaks, shifts etc. These objects also inherit from `SimPy.Process` class.

The most important attribute of an `ObjectInterruption` is *victim* which is the `CoreObject` whose the availability the `ObjectInterruption` handles. This `CoreObject` is also the one that creates and activates the instance of the `ObjectInterruption` object.

Currently there are three generic methods for these objects:

- *outPutTrace*: outputs trace in an Excel sheet when an important event happens (e.g. a Machine gets a failure). All the `ObjectInterruptions` output to the same Excel file as the `CoreObjects` and the events are sorted in increasing timestamp. The trace is essential for debugging. To run a model that is believed to be verified, it should be turned off since it slows the program significantly.
- *getVictimQueue*: returns the internal queue of the victim `CoreObject`.
- *run*: this is a generator method and it is the only one where the *yield* commands of SimPy can be used. For this reason *run* requires that the user knows the internals of SimPy in order to customize. Generally the victim `CoreObject` is the one that activates the `ObjectInterruption`, but this is not obligatory. It is common (but not obligatory) that in such a method there is a *while* loop that runs all through the simulated time. The logic followed in an `ObjectInterruption`'s *run* method is:
  1. Hold until an interruption should happen
  2. Passivate the victim
  3. Hold until the interruption should be stopped
  4. Reactivate the victim
  5. Restart the start of the loop

### 3.1.3 Entity

As Entities are categorized all objects that get processed by or wait in CoreObjects and they are not permanent in a model. For example parts in a production line, customers in a shop, calls in a call centre etc.

Entities can get into the model from a Source type CoreObject or be set as Work In Progress (WIP) at the start of the simulation run. They hold certain general attributes such as *creationTime* that holds the time that the Entity entered the model.

ManPy Entities do NOT inherit from SimPy.Process class. This is in alignment with the philosophy of having the CoreObjects handling most of the simulation logic. So the CoreObjects handle how the Entities move and evolve through simulated time. Of course it is possible that certain properties (such as routing or processing time needed) may be kept in an Entity's attributes, which the CoreObject will read.

Currently there are two generic methods for these objects:

- *outPutResultsJSON*: outputs the results of the object in a JSON format. All the Entities output to the same JSON file as the CoreObjects.
- *initialize*: initializes the Entity at the start of each replication.

### 3.1.4 ObjectResource

As ObjectResource are categorized all the resources that might be necessary for certain operation of a CoreObject. For example repairman, operator, electric power etc. An ObjectResource is necessary in modelling when two or more CoreObjects compete for the same resource (e.g. two machines competing for the same operator).

ManPy ObjectResources do NOT inherit from SimPy.Process class. This is in alignment with the philosophy of having the CoreObjects handling most of the simulation logic. So the CoreObjects handle how the ObjectResources move and evolve through simulated time. Of course it is possible that certain properties may be kept in an ObjectResource's attributes, which the CoreObject will read.

One important attribute if the ObjectResource is *Res*. *Res* is an instance SimPy.resource type and it allows other objects to request or release the resource (SimPy *yield.request* and *yield.release* respectively).

- *postProcessing*: Same functionality with CoreObject method with the same name
- *outPutResultsJSON*: Same functionality with CoreObject method with the same name
- *outPutResultsXL*: Same functionality with CoreObject method with the same name
- *outPutTrace*: Same functionality with CoreObject method with the same name
- *initialize*: Same functionality with CoreObject method with the same name
- *checkIfResourcesAvailable*: returns true if there is one or more available units of the ObjectResource.
- *getResource*: returns the resource (*self.Res*)
- *getResourceQueue*: returns the activeQueue of the resource (*self.Res.activeQueue*)

### 3.1.5 Auxiliary

These are auxiliary classes that are needed for different simulation functionalities. Unlike the other categories described here, auxiliary classes do not inherit from one parent class, even though it is considered a good practice that they are grouped and presented here for reasons of coherence.

Three categories of auxiliary classes exist currently in ManPy.

- *G*: contains global variables for the simulation such as the length of the simulation run, the number of the simulation replications etc. *G* can be imported with the line *from Globals import G*. Some important conventions:

- G.ObjList is a list that should hold all the CoreObjects.
- G.maxSimTime is a float that defines the length of the simulation run.
- G.seed is an integer that holds the seed for random number generation.
- *RandomNumberGenerator*: contains methods to create random variables that follow certain distributions. In the current version of ManPy only a few distributions listed below are supported, but this is to be expanded:
  - Fixed
  - Exponential
  - Normal
  - Erlang
- *MainScript*: as main script we name every script (it is not necessary a class) that reads a ManPy simulation model, creates it, runs it and returns the results. The input and output can be of whatever form. There are currently two different main scripts *LineGenerationJSON* and *LineGenerationCMSD* that read the data using different formats. Also, all the examples demonstrated in the next section are main scripts. Nevertheless, it is desired that users can implement and use different main scripts according to their needs. A main script should perform the following operations:
  1. Read or define the objects
  2. Create the objects
  3. Define the structure and set the topology of the model (predecessors and successors) if needed
  4. In every replication:
    - i. initialize the simulation (SimPy.initialize)
    - ii. initialize CoreObjects, ObjectResources and Entities
    - iii. set the WIP if needed
    - iv. activate the objects
    - v. run the simulation (SimPy.simulate)
    - vi. call postProcessing method of the objects
  5. After the simulation is over output the results in a desirable way

## 3.2 Expanding the Code

In the last subsection the architecture, generic methods and the logic of ManPy were described. Understanding the above, it should be possible for someone to make a new object of any of the 5 categories described and incorporate it into the platform. New objects may be:

- Customized objects that inherit from an existing one and override certain methods
- Completely new objects, that implement their versions of the methods

In order to reduce the learning curve, it is desired that ManPy keeps the set of methods as short as possible. However, adding a new eneric method in a new object is also possible. Let's suppose for example that a CoreObject named *newCoreObject* requires having *newCoreObjectMethod* that will also be called by other objects in the model. Then the developer can implement the version for *newCoreObjectMethod* that he wishes for the *newCoreObject*, but he should also add an empty version of the method to the parent object. So in CoreObject the following should be added:

```
def newCoreObjectMethod(self):
    pass
```

In case *newCoreObjectMethod* requires arguments, they should be defined as optional. In this way the method can be called for every CoreObject without causing the code to crash and the objects can still interact as black boxes.

In the next section examples of how to construct, customize and run a ManPy model will be given.

## 4 Examples

### 4.1 A single server model

The first example shown here is a simple model of a production line that consists only from a point of entry (Source) one server (Machine) and a point of exit (Exit). A graphical representation of the model is shown in Figure2 (Note, Figure2 and other figures in this section, are printscreens from the DREAM GUI. They are presented here for convenience, in order to make the text more understandable. This documentation is specific for ManPy and does NOT cover the DREAM GUI).

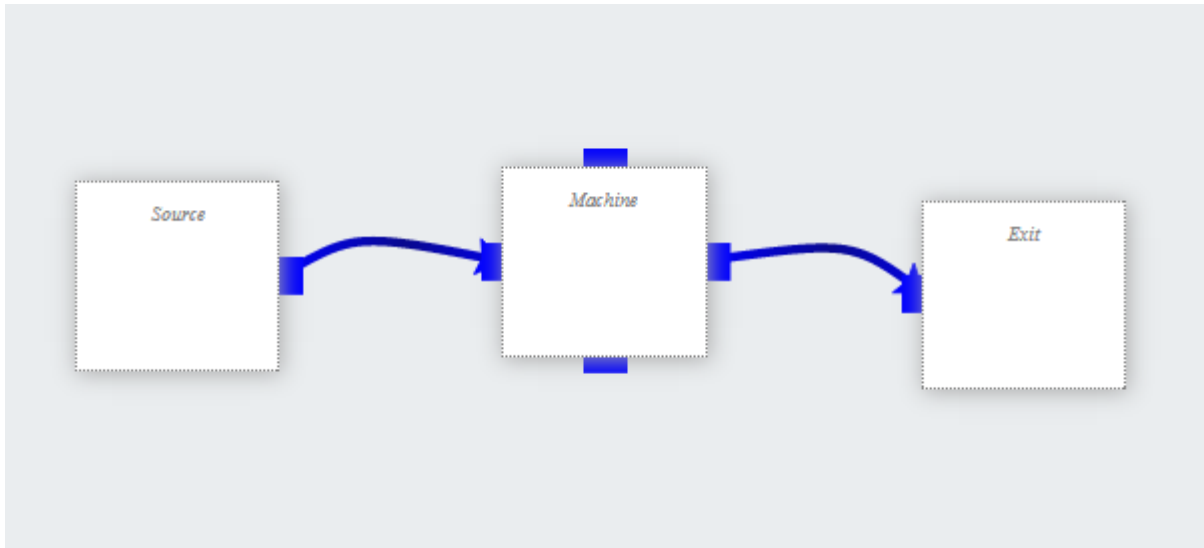


Figure2: Single server model

As values we have the following:

- The source produces parts. One part is produced every 30 seconds
- The Machine processes one part at a time. The processing time is 15 seconds
- We want to study the system for 24 hours

Below is the ManPy main script to run this model (dream\simulation\Examples\SingleServer.py):

```
from SimPy.Simulation import simulate, activate, initialize
from simulation.Machine import Machine
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulationGlobals import G

#define the objects of the model
S=Source('S1','Source',distribution='Fixed', mean=0.5, item=Part)
M=Machine('M1','Machine', mean=0.25)
E=Exit('E1','Exit')

G.ObjList=[S,M,E] #add all the objects in G.ObjList so that they can be easier
accessed later

#define predecessors and successors for the objects
S.defineRouting(successorList=[M])
M.defineRouting(predecessorList=[S],successorList=[E])
E.defineRouting(predecessorList=[M])
```

```

initialize()                                #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

G.maxSimTime=1440.0    #set G.maxSimTime 1440.0 minutes (1 day)

simulate(until=G.maxSimTime)    #run the simulation

#carry on the post processing operations for every object in the topology
for object in G.ObjList:
    object.postProcessing()

#print the results
print "the system produced", E.numOfExits, "parts"
print "the total working ratio of the Machine is",
(M.totalWorkingTime/G.maxSimTime)*100, "%"

```

Running the model we get the following in our console:

```

the system produced 2880 parts
the working ratio of the Machine is 50.0 %

```

Some notes on the code:

- ManPy needs and abstract time unit. The user defines what this is. In this model we picked minutes. The length of the simulation is set to 1440 minutes (24 hours).
- ManPy time units are decimals. So 30 and 15 seconds have been translated to 0.5 and 0.25 minutes respectively.
- *defineRouting* in most CoreObjects gets two lists as arguments (perdecessorList, successorList) with this sequence. In special cases like the Source and the Exit only one list is required. In this example the name of the argument is specified when the method is called, but if the user gives the inputs with the same sequence (see next examples) the result shall be the same.
- We see that in the Source the distribution is given as argument 'Fixed', while in the Machine we did not do the same. 'Fixed' is the default ManPy distribution, so it is not needed.

We see the results are logical:

- In 1440 minutes and a part coming every 0.5 minutes and staying in the system 0.25 minutes (no blocking) it is normal to produce 2880 parts
- Since parts come every 30 seconds and the machine processes them for 15 seconds it is logical

(Note: all the programs presented here, and generally ManPy objects are verified against a commercial simulation package. We use Plant Simulation - [http://www.plm.automation.siemens.com/en\\_us/products/tecnomatix/plant\\_design/plant\\_simulation.shtml](http://www.plm.automation.siemens.com/en_us/products/tecnomatix/plant_design/plant_simulation.shtml))

## 4.2 Two servers model with failures and repairman

The second model is a bit more complex. The graphical representation is available in Figure3

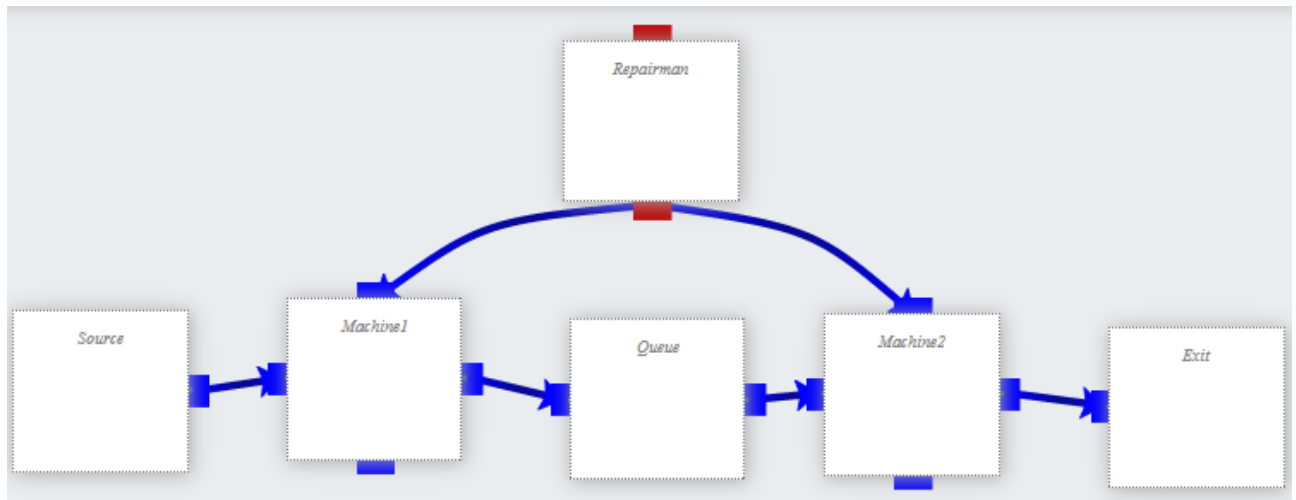


Figure4: Two servers model with failures and repairman

In this model we have two Machines and a Queue between them. The Machines are vulnerable to failures and when a failure happens then they need a repairman to get fixed. In our model there is only one repairman named Bob available. We have the following data:

- The source produces parts. One part is produced every 30 seconds
- For Machine1
  - Processing time is Fixed to 15 seconds
  - MTTF is 1 hour
  - MTTR is 5 minutes
- For Machine2
  - Processing time is Fixed to 90 seconds
  - MTTF is 40 minutes
  - MTTR is 10 minutes
- The capacity of the Queue is 1
- We want to study the system in a 24 hours period and identify the number of items that are produced, the blockage ratio in Machine1 and the working ration of the repairman.

Below is the ManPy main script to run this model (dream\simulation\Examples\TwoServers.py):

```
from SimPy.Simulation import simulate, activate, initialize
from simulation.Machine import Machine
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulation.Repairman import Repairman
from simulation.Queue import Queue
from simulation.Globals import G

#define the objects of the model
R=Repairman('R1', 'Bob')
S=Source('S1', 'Source', mean=0.5, item=Part)
M1=Machine('M1', 'Machine1', mean=0.25, failureDistribution='Fixed', MTTF=60, MTTR=5,
repairman=R)
Q=Queue('Q1', 'Queue')
M2=Machine('M2', 'Machine2', mean=1.5, failureDistribution='Fixed', MTTF=40,
MTTR=10, repairman=R)
E=Exit('E1', 'Exit')
```

```

G.ObjList=[S,M1,M2,E,Q]    #add all the objects in G.ObjList so that they can be easier
                             accessed later

#define predecessors and successors for the objects
S.defineRouting([M1])
M1.defineRouting([S],[Q])
Q.defineRouting([M1],[M2])
M2.defineRouting([Q],[E])
E.defineRouting([M2])

initialize()                #initialize the simulation (SimPy method)

#initialize all the objects
R.initialize()

for object in G.ObjList:
    object.initialize()

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

G.maxSimTime=1440.0        #set G.maxSimTime 1440.0 minutes (1 day)

simulate(until=G.maxSimTime)    #run the simulation

#carry on the post processing operations for every object in the topology
for object in G.ObjList:
    object.postProcessing()
R.postProcessing()

#print the results
print "the system produced", E.numOfExits, "parts"
print "the blockage ratio of", M1.objName, "is",
(M1.totalBlockageTime/G.maxSimTime)*100, "%"
print "the working ratio of", R.objName,"is", (R.totalWorkingTime/G.maxSimTime)*100,
"%"

```

Running the model we get the following in our console:

```

the system produced 732 parts
the blockage ratio of Machine1 is 78.1770833333 %
the working ratio of Bob is 26.7361111111 %

```

Some notes:

- It is handy to declare the Repairman first so that the machines take him as argument
- For the Repairman the *initialize* and *postProcessing* methods should also be ran, but not the *activate* (Repairman is in the category of ObjectResource that is not a SimPy.process)



### 4.3 An assembly line

In this example we use another ManPy object. Assembly takes two types of Entities, parts and frames. A frame can be loaded with a number of parts. The logic is that the Assembly waits first for a frame and when it has one then it loads the parts to it when they arrive.

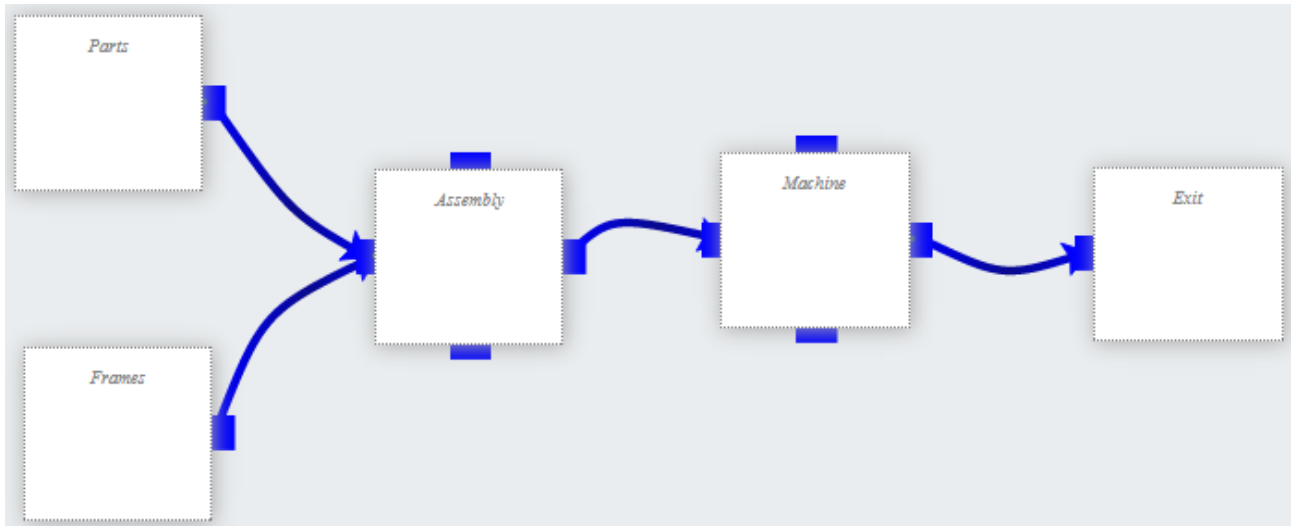


Figure4: An assembly line

- “Parts” produces parts. One part is produced every 30 seconds
- “Frames” produces parts. One frame is produced every 2 minutes
- A Frame has a fixed capacity of 4 parts
- The Assembly has a fixed processing time of 2 minutes
- For Machine
  - Processing time is Fixed to 15 seconds
  - MTTF is 1 hour
  - MTTR is 5 minutes
- We want to study the system in 24 hours and identify the number of items that are produced and the blockage ratio in Assembly.

Below is the ManPy main script to run this model (dream\simulation\Examples\AssemblyLine.py):

```
from SimPy.Simulation import simulate, activate, initialize
from simulation.Machine import Machine
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulation.Frame import Frame
from simulation.Assembly import Assembly
from simulation.Globals import G

#define the objects of the model
Frame.capacity=4
Sp=Source('SP','Parts', mean=0.5, item=Part)
Sf=Source('SF','Frames', mean=2, item=Frame)
M=Machine('M','Machine', mean=0.25, failureDistribution='Fixed', MTTF=60, MTTR=5)
A=Assembly('A','Assembly', mean=2)
E=Exit('E1','Exit')
```

```

G.ObjList=[Sp,Sf,M,A,E]    #add all the objects in G.ObjList so that they can be easier
                             accessed later

#define predecessors and successors for the objects
Sp.defineRouting([A])
Sf.defineRouting([A])
A.defineRouting([Sp,Sf],[M])
M.defineRouting([A],[E])
E.defineRouting([M])

initialize()                #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

G.maxSimTime=1440.0        #set G.maxSimTime 1440.0 minutes (1 day)

simulate(until=G.maxSimTime)    #run the simulation

#carry on the post processing operations for every object in the topology
for object in G.ObjList:
    object.postProcessing()

#print the results
print "the system produced", E.numOfExits, "frames"
print "the working ratio of", A.objName, "is", (A.totalWorkingTime/G.maxSimTime)*100,
"%"

```

Running the model we get the following in our console:

```

the system produced 664 frames
the working ratio of Assembly is 92.3611111111 %

```

Note that the capacity of the frames is set as an attribute of the class with *Frame.capacity=4*

## 4.4 Parallel stations and Queue customization

In this example we will see some deeper aspects of customizing an object. Our model consists of a source, a buffer and two Milling machines that work in parallel. A graphical representation is given in Figure5. We have the following data:

- The source produces parts. One part is produced every 30 seconds
- For Machine1
  - Processing time is Fixed to 15 seconds
  - MTTF is 1 hour
  - MTTR is 5 minutes
- For Machine2
  - Processing time is Fixed to 15 seconds

- No failures
- The capacity of the Queue is infinite
- We want to study the system in 24 hours and identify the number of items that are produced, the working ratio of both Machines

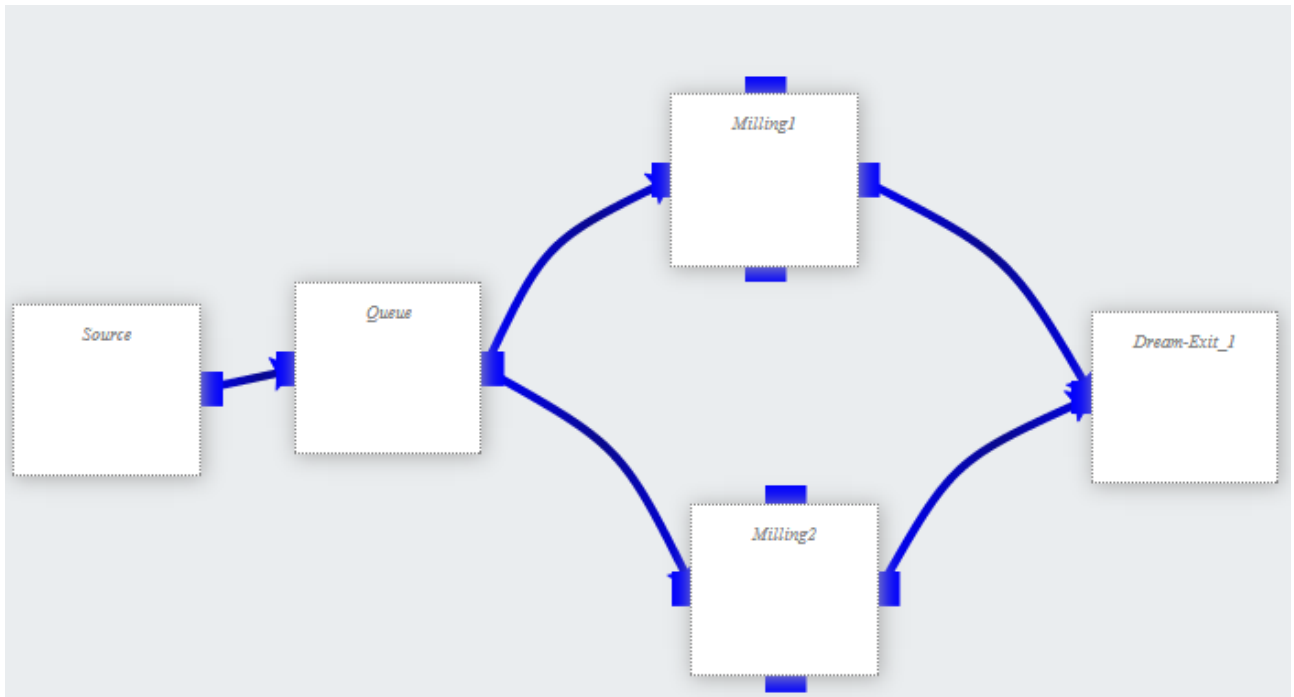


Figure5: Parallel stations and Queue customization

To model this scenario we need nothing more than we already described. The code is given below (dream\simulation\Examples\ParallelServers1.py)

```

from SimPy.Simulation import simulate, activate, initialize, infinity
from simulation.Machine import Machine
from simulation.Queue import Queue
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulation.Globals import G

#define the objects of the model
S=Source('S','Source', mean=0.5, item=Part)
Q=Queue('Q','Queue', capacity=infinity)
M1=Machine('M1','Milling1', mean=0.25, failureDistribution='Fixed', MTTF=60, MTTR=5)
M2=Machine('M2','Milling2', mean=0.25)
E=Exit('E1','Exit')

G.ObjList=[S,Q,M1,M2,E] #add all the objects in G.ObjList so that they can be easier
accessed later

#define predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[M1,M2])
M1.defineRouting([Q],[E])
M2.defineRouting([Q],[E])
E.defineRouting([M1,M2])

```

```

initialize()                                #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

G.maxSimTime=1440.0    #set G.maxSimTime 1440.0 minutes (1 day)

simulate(until=G.maxSimTime)    #run the simulation

#carry on the post processing operations for every object in the topology
for object in G.ObjList:
    object.postProcessing()

#print the results
print "the system produced", E.numOfExits, "parts"
print "the working ratio of", M1.objName, "is",
(M1.totalWorkingTime/G.maxSimTime)*100, "%"
print "the working ratio of", M2.objName, "is",
(M2.totalWorkingTime/G.maxSimTime)*100, "%"

```

Running the model we get the following in our console:

```

the system produced 2880 parts
the working ratio of Milling1 is 23.0902777778 %
the working ratio of Milling2 is 26.9097222222 %

```

We see that Milling2 is slightly busier than Milling1. This is logical since Milling1 gets also failures. Let's assume now, that in our real system, Milling1 has a greater priority than Milling2, i.e a part will go to Milling1, unless it is not available so it will go to Milling2.

The default behaviour of Queue is to handle things in a cyclic way (if both successors available select first Milling1 then Milling2 etc). To change this we have to override Queue's *haveToDispose* method.

The code is given below (dream\simulation\Examples\ParallelServers2.py).

```

from SimPy.Simulation import simulate, activate, initialize, infinity
from simulation.Machine import Machine
from simulation.Queue import Queue
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulation.Globals import G

#the custom queue
class SelectiveQueue(Queue):
    def haveToDispose(self, callerObject=None):
        caller=callerObject
        #if the caller is Milling1 then we return true if there are parts queued
        if caller.id=='M1':

```

```

        return len(self.getActiveObjectQueue())>0
    #if the caller is Milling2 then we have to check Milling1's condition.
    #we return true if there are parts queued AND Milling1 cannot accept them
    self.M1=None
    if caller.id=='M2':
        #loop through the objects to identify Milling1
        for object in G.ObjList:
            if object.id=='M1':
                self.M1=object
        #check Queue's status and also if Milling1 can accept
        return len(self.getActiveObjectQueue())>0 and (not (self.M1.canAccept()))

#define the objects of the model
S=Source('S','Source', mean=0.5, item=Part)
Q=SelectiveQueue('Q','Queue', capacity=infinity)    #Q is now of type SelectiveQueue
M1=Machine('M1','Milling1', mean=0.25, failureDistribution='Fixed', MTTF=60, MTTR=5)
M2=Machine('M2','Milling2', mean=0.25)
E=Exit('E1','Exit')

G.ObjList=[S,Q,M1,M2,E]    #add all the objects in G.ObjList so that they can be easier
                             accessed later

#define predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[M1,M2])
M1.defineRouting([Q],[E])
M2.defineRouting([Q],[E])
E.defineRouting([M1,M2])

initialize()                #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

G.maxSimTime=1440.0        #set G.maxSimTime 1440.0 minutes (1 day)

simulate(until=G.maxSimTime)    #run the simulation

#carry on the post processing operations for every object in the topology
for object in G.ObjList:
    object.postProcessing()

#print the results
print "the system produced", E.numOfExits, "parts"
print "the working ratio of", M1.objName, "is",
(M1.totalWorkingTime/G.maxSimTime)*100, "%"
print "the working ratio of", M2.objName, "is",
(M2.totalWorkingTime/G.maxSimTime)*100, "%"

```

Running the model we get the following in our console:

```

the system produced 2880 parts
the working ratio of Milling1 is 46.1805555556 %
the working ratio of Milling2 is 3.81944444444 %

```

We see now that the working ration of Milling2 is drastically reduced that is natural since it takes parts only when Milling1 is busy or failed.

Some notes on the code:

- SelectiveQueue is a new custom object. It has its own version of haveToDispose, but in everything else it is identical to Queue.
- Q is now of type SelectiveQueue
- The implementation of SelectiveQueue is highly custom. It works only in this model with the given ids ('M1' and 'M2'). A more generic NewQueue could be made. Users are welcome to customize their objects at different levels:
  - Objects for specific models like the SelectiveQueue shown
  - More generic objects so that the user can re-use the in different models
  - Even more generic objects so that the user can share them with other users. Here documentation would be essential.

## 4.5 Parallel stations and counting the parts of each machine

In the previous example, we assume that in the Exit we want to count how many parts were processed by Milling1 and how many by Milling2. For this we need to make 3 modifications:

- Create two new global variables:
  - G.NumM1 as a counter that counts the parts that were processed by Milling1
  - G.NumM2 as a counter that counts the parts that were processed by Milling2
- Create a new Machine type named Milling. This will override the getEntity method so that it sets an attribute to the part that shows from which Milling it passed
- Create a new Exit type named CountingExit. This will override the getEntity method so that it reads the attribute of the part and increments the global counters accordingly

The code is given below (dream\simulation\Examples\ParallelServers3.py).

```
from SimPy.Simulation import simulate, activate, initialize, infinity, now
from simulation.Machine import Machine
from simulation.Queue import Queue
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulation.Globals import G

#the custom queue
class SelectiveQueue(Queue):
    def haveToDispose(self, callerObject=None):
        caller=callerObject
        if caller.id=='M1':
            return len(self.getActiveObjectQueue())>0
        self.M1=None
        if caller.id=='M2':
            for object in G.ObjList:
                if object.id=='M1':
                    self.M1=object
            return len(self.getActiveObjectQueue())>0 and (not (self.M1.canAccept()))

#the custom machine
class Milling(Machine):
    def getEntity(self):
```

```

        Machine.getEntity(self)          #call the parent method to get the entity
        part=self.getActiveObjectQueue()[0] #retrieve the obtained part
        part.machineId=self.id           #create an attribute to the obtained part
and give it the value of the object's id

#the custom exit
class CountingExit(Exit):
    def getEntity(self):
        part=self.getGiverObjectQueue()[0] #find the part to be obtained
        Exit.getEntity(self)               #call the parent method to get the
entity
        #check the attribute and update the counters accordingly
        if part.machineId=='M1':
            G.NumM1+=1
        elif part.machineId=='M2':
            G.NumM2+=1

#define the objects of the model
S=Source('S','Source', mean=0.5, item=Part)
Q=SelectiveQueue('Q','Queue', capacity=infinity)
M1=Milling('M1','Milling1', mean=0.25, failureDistribution='Fixed', MTTF=60, MTTR=5)
M2=Milling('M2','Milling2', mean=0.25)
E=CountingExit('E1','Exit')

G.ObjList=[S,Q,M1,M2,E] #add all the objects in G.ObjList so that they can be easier
accessed later

#create the global variables
G.NumM1=0
G.NumM2=0

#define predecessors and successors for the objects
S.defineRouting([Q])
Q.defineRouting([S],[M1,M2])
M1.defineRouting([Q],[E])
M2.defineRouting([Q],[E])
E.defineRouting([M1,M2])

initialize() #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

G.maxSimTime=1440.0 #set G.maxSimTime 1440.0 minutes (1 day)

simulate(until=G.maxSimTime) #run the simulation

#carry on the post processing operations for every object in the topology
for object in G.ObjList:
    object.postProcessing()

#print the results
print "the system produced", E.numOfExits, "parts"

```

```

print "the working ratio of", M1.objName, "is",
(M1.totalWorkingTime/G.maxSimTime)*100, "%"
print "the working ratio of", M2.objName, "is",
(M2.totalWorkingTime/G.maxSimTime)*100, "%"
print M1.objName, "produced", G.NumM1, "parts"
print M2.objName, "produced", G.NumM2, "parts"

```

Running the model we get the following in our console:

```

the system produced 2880 parts
the working ratio of Milling1 is 46.1805555556 %
the working ratio of Milling2 is 3.81944444444 %
Milling1 produced 2660 parts
Milling2 produced 220 parts

```

## 4.6 Stochastic model

All the models so far have been deterministic. Real systems tend to be random with different reasons of stochasticity. In stochastic models we have to run many replications with different random seeds and give the results in confidence intervals.

We take our second example (dream\simulation\Examples\TwoServers.py) and we extend it into a stochastic situation. The model is the same, the only change is that the machines have stochastic processing times. More specifically:

- Machine1 processing time follows the normal distribution with mean=0.25, stdev=0.1, min=0.1, max=1 (all in minutes)
- Machine2 processing time follows the normal distribution with mean=1.5, stdev=0.3, min=0.5, max=5 (all in minutes)

The failures and the interarrival times remain deterministic as before.

Below is the ManPy main script to run this model (dream\simulation\Examples\TwoServersStochastic.py):

```

from SimPy.Simulation import simulate, activate, initialize
from simulation.Machine import Machine
from simulation.Source import Source
from simulation.Exit import Exit
from simulation.Part import Part
from simulation.Repairman import Repairman
from simulation.Queue import Queue
from simulation.Globals import G

#define the objects of the model
R=Repairman('R1', 'Bob')
S=Source('S1','Source', mean=0.5, item=Part)
M1=Machine('M1','Machine1', distribution='Normal', mean=0.25, stdev=0.1, min=0.1,
max=1, failureDistribution='Fixed', MTTF=60, MTTR=5, repairman=R)
Q=Queue('Q1','Queue')
M2=Machine('M2','Machine2', distribution='Normal', mean=1.5, stdev=0.3, min=0.5, max=5,
failureDistribution='Fixed', MTTF=40, MTTR=10,repairman=R)
E=Exit('E1','Exit')

G.ObjList=[S,M1,M2,E,Q] #add all the objects in G.ObjList so that they can be easier
accessed later

#define predecessors and successors for the objects
S.defineRouting([M1])

```



```

M1.defineRouting([S],[Q])
Q.defineRouting([M1],[M2])
M2.defineRouting([Q],[E])
E.defineRouting([M2])

G.maxSimTime=1440.0      #set G.maxSimTime 1440.0 minutes (1 day)
G.numberOfReplications=10  #set 10 replications
G.confidenceLevel=0.99    #set the confidence level. 0.99=99%

#run the replications
for i in range(G.numberOfReplications):
    G.seed+=1            #increment the seed so that we get different random numbers in each
    run.

    initialize()          #initialize the simulation (SimPy method)

    #initialize all the objects
    R.initialize()
    for object in G.ObjList:
        object.initialize()

    #activate all the objects
    for object in G.ObjList:
        activate(object, object.run())
    simulate(until=G.maxSimTime)    #run the simulation

    #carry on the post processing operations for every object in the topology
    for object in G.ObjList:
        object.postProcessing()
    R.postProcessing()

#output data to excel for every object
for object in G.ObjList:
    object.outputResultsXL()
R.outputResultsXL()

G.outputFile.save("output.xls")

```

Running this we get in output.xls the following:

CI 99.0% for the mean percentage of Failure of Machine1 is:	9,027778	9,027778	9,027778
CI 99.0% for the mean percentage of Working of Machine1 is:	13,29828	13,47811	13,65793
CI 99.0% for the mean percentage of Blockage of Machine1 is:	77,28284	77,46229	77,64174
CI 99.0% for the mean percentage of Waiting of Machine1 is:	0,025811	0,031827	0,037843
CI 99.0% for the mean percentage of Failure of Machine2 is:	19,79167	19,79167	19,79167
CI 99.0% for the mean percentage of Working of Machine2 is:	76,22322	76,40801	76,5928
CI 99.0% for the mean percentage of Blockage of Machine2 is:	-2,5E-14	2,08E-14	6,68E-14
CI 99.0% for the mean percentage of	3,615533	3,800321	3,985109

Waiting of Machine2 is:

CI 99.0% for the mean Throughput in Exit is:	727,3562	733,9	740,4438
CI 99.0% for the mean Lifespan of an entity that exited from Exit is:	533,2338	535,4324	537,631
CI 99.0% for the avg takt time in Exit is:	1,968579	1,968579	1,968579
CI 99.0% for the mean percentage of Working of Bob is:	26,73611	26,73611	26,73611
CI 99.0% for the mean percentage of Waiting of Bob is:	73,26389	73,26389	73,26389

Some notes:

- This main script needs a loop where all the replications are ran. In every replication the random number seed should be changed and also the objects initialized and activated again.
- The results presented here are the standard results the CoreObjects of every type output. The user can customize this by overriding the *outputResultsXL* method of an object.
- In order to calculate confidence intervals ManPy uses SciPy (<http://www.scipy.org/>). The user can obtain the outcome of every run individually and apply his own analysis.
- G.outputFile is a ManPy global to define the output excel file.
- In the normal distribution it is on the developer's responsibility not to give irrational values. For example if a processing time is negative ManPy will crash, Another example if min is larger than max in normal distribution, ManPy would try to create these numbers and it will run forever.
- Some comments on the results:
  - In some results all 3 numbers are the same. This is because failures are still deterministic and they are not affected by stochasticity. The same applies for the Repairman.
  - The blockage of Machine2 is logically 0 (since it sends to an Exit that is always available). Sometimes ManPy can give very small values like in this example. They should be considered as 0.
  - In small values, for example the waiting portion of a machine that is almost always busy, the left bound of the confidence interval can become negative even if it is not logical (we cannot have negative waiting). Such values the user can set to 0.

## 4.7 A simple Job-Shop

So far all the CoreObjects had dedicated predecessors and successors. There are situations where it is desirable to model a job shop system where CoreObjects can give/receive to/from whichever other CoreObject in the model. The information of which CoreObject is the next station is an attribute of the Entity. Also, there are situations where we have hybrid systems (there are some predecessor-successor associations, but also job-shop logic). As an example we give the model of Figure 6. In this model, the Machines have dedicated input buffers and the Queues dedicated output stations. For example, Q1 can give only to M1 and M1 can receive only from Q1. On the other hand Q1 can receive from all the machines and M1 can give to all the queues or the Exit.

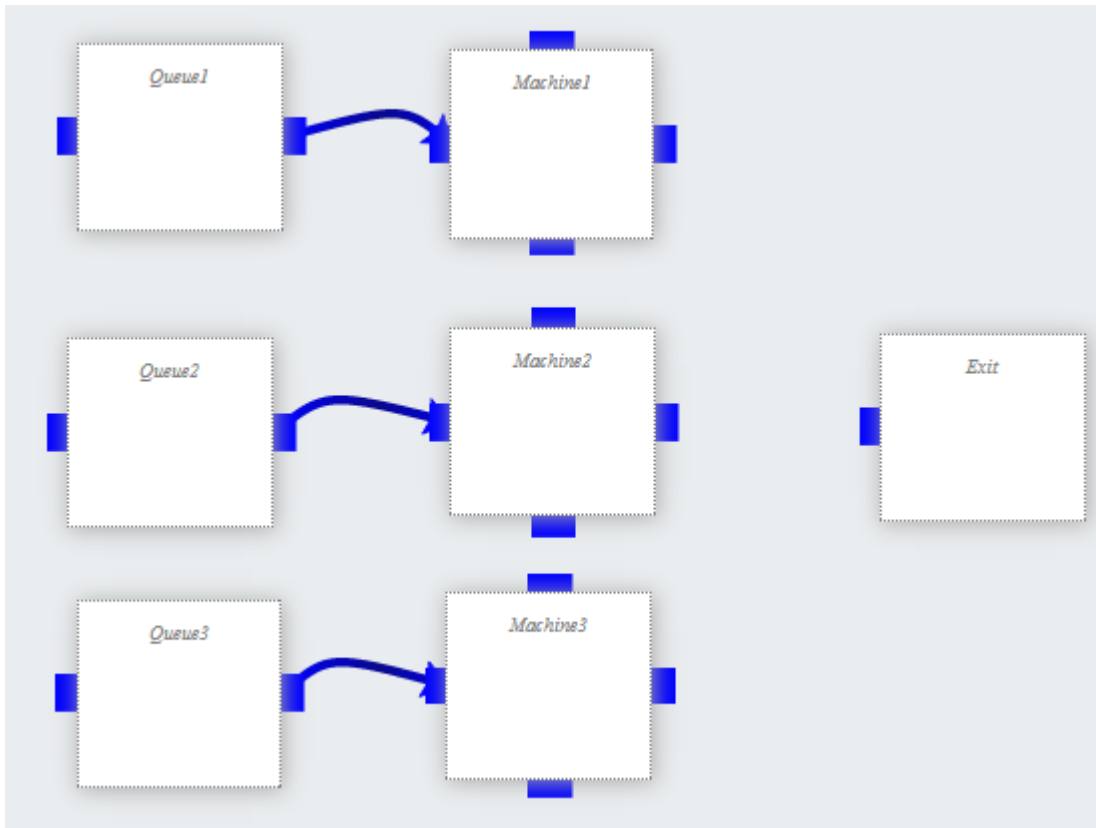


Figure 6: a simple job shop

To model such situations ManPy object repository has the following objects:

- **MachineJobShop**: inherits from Machine and overrides the logic of methods such as *getEntity* or *haveToDispose* in order to be able to give in every CoreObject. Also it overrides *getReceiverObject* in order to identify the receiver from the Entity's attributes and *calculateProcessingTime* in order to calculate the processing time according to the Entity's attributes.
- **QueueJobShop**: inherits from Queue but overrides the logic of methods such as *getEntity* or *canAccept* in order to be able to receive from every CoreObject. Also it overrides *getGiverObject* in order to identify the giver from the Entity's attributes
- **ExitJobShop**: inherits from Exit but overrides the logic of methods such as *getEntity* or *canAccept* in order to be able to receive from every CoreObject. Also it overrides *getGiverObject* in order to identify the giver from the Entity's attributes
- **Job**: inherits from Entity. One of its attributes is a list named *route*. This list has the following form `[[id1,processingTime1], [id2,processingTime2], ..., [idN,processingTimeN]]`. Every item in *route* corresponds to the id of a CoreObject and the processing time in this CoreObject. If the CoreObject has a standard input (like the Machines in our example) it should not be in *route*. Another attribute called *remainingRoute* is also a list that holds the future stops of a Job at any moment of simulation time. In the beginning of the simulation these lists are equal. Current implementation of Job can be used only for Fixed processing times. Job has also a list named *schedule*, which is updated by the CoreObject every time it receives the Job. This holds the output for the Job, i.e. which stations it entered and when.

In our first simple example we assume that we have only one Job in the model shown in Figure 6. Our data for this Job is:

- It starts in Queue1 and it has to visit Machine1, Machine3 and Machine2 (in this sequence) before it exits the system
- Its processing time in M1 is 1

- Its processing time in M3 is 3
- Its processing time in M2 is 2

Below is the ManPy main script to run this model (dream\simulation\Examples\JobShop1.py):

```
from SimPy.Simulation import simulate, activate, initialize, infinity, now
from simulation.MachineJobShop import MachineJobShop
from simulation.QueueJobShop import QueueJobShop
from simulation.ExitJobShop import ExitJobShop
from simulation.Job import Job
from simulation.Globals import G

#define the objects of the model
Q1=QueueJobShop('Q1','Queue1', capacity=infinity)
Q2=QueueJobShop('Q2','Queue2', capacity=infinity)
Q3=QueueJobShop('Q3','Queue3', capacity=infinity)
M1=MachineJobShop('M1','Machine1')
M2=MachineJobShop('M2','Machine2')
M3=MachineJobShop('M3','Machine3')
E=ExitJobShop('E','Exit')

G.ObjList=[M1,M2,M3,Q1,Q2,Q3,E] #add all the objects in G.ObjList so that they can be
easier accessed later

#define predecessors and successors for the objects
Q1.defineRouting(successorList=[M1])
Q2.defineRouting(successorList=[M2])
Q3.defineRouting(successorList=[M3])
M1.defineRouting(predecessorList=[Q1])
M2.defineRouting(predecessorList=[Q2])
M3.defineRouting(predecessorList=[Q3])

#define the Jobs
J=Job('J1','Job1',route=[[ 'Q1',1],[ 'Q3',3],[ 'Q2',2],[ 'E',0]])

initialize() #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()
J.initialize()

#set the WIP
Q1.getActiveObjectQueue().append(J) #place the Job at 'Q1'
J.remainingRoute[0][0]='' #remove data from the remaining route since it is
already added in Q1.
#this is to make sure that the Job will not get again
into Queue1 while it waits in Queue1
J.schedule.append(['Q1',now()]) #add the data in the schedule that the Job entered Q1
at time=0

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

simulate(until=infinity) #run the simulation until there are no more events

G.maxSimTime=E.timeLastEntityLeft #calculate the maxSimTime as the time that the last
Job left
```

```
#loop in the schedule to print the results
for record in J.schedule:
    #schedule holds ids of objects. The following loop will identify the name of the
    CoreObject with the given id
    name=None
    for obj in G.ObjList:
        if obj.id==record[0]:
            name=obj.objName
    print J.name, "got into", name, "at", record[1]
```

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 0
Job1 got into Queue3 at 1
Job1 got into Machine3 at 1
Job1 got into Queue2 at 4
Job1 got into Machine2 at 4
Job1 got into Exit at 6
```

Having only one Job it is very easy to confirm that we got the correct result.

Some notes on the code:

- MachineJobShop and QueueJobShop are custom objects. MachineJobShop acts as job shop only when it gives and QueueJobShop only when it receives. A more generic approach is to be developed.
- In such situations it is handy to define predecessor and successor lists explicitly in the CoreObjects *defineRouting*.
- We see that the *route* of the Job is given with the following argument `[[Q1',1],[Q3',3],[Q2',2],[E',0]]`. So the Machines are not written but in the Queues the processing time of the Machines is applied. The justification is as follows:
  - Machines have dedicated input (predecessors) so they should not be in the route
  - Queues are passive, so they have no processing time by default. So this processing time is applied to the dedicated successor.
  - The Exit is assigned with processing time=0
  - This is a “dirty” implementation of Job, a more generic one is to be developed.
- This situation is different than the ones we have seen before because we have no Source that creates Entities. The Job has to be set as WIP.
- Also, we do not have a predefined length of simulation; it will end when the Job exits the system. For this reason we run the simulation for infinite time, and SimPy.simulate will stop when there are no more events. Notice that if a user applies infinite time in a simulation that does not stop to produce events (like in the previous examples), then the execution of the model will never stop. With the line `G.maxSimTime=E.timeLastEntityLeft` after the execution of the simulation we can calculate the length of the simulation in case it is needed for future results (e.g. to calculate the working ratio of a Machine)

## 4.8 A Job-Shop with scheduling rules

For the model described in the previous example and in Figure 6 we assume now that we have 3 Jobs. For these Jobs we know:

- Job1:
  - It starts in Queue1 and it has to visit Machine1, Machine3 and Machine2 (in this sequence) before it exits the system
  - Its processing time in M1 is 1
  - Its processing time in M3 is 3
  - Its processing time in M2 is 2
  - Its priority is 1
  - Its due date is 100
- Job2:
  - It starts in Queue1 and it has to visit Machine1, Machine2 and Machine3 (in this sequence) before it exits the system
  - Its processing time in M1 is 2
  - Its processing time in M2 is 4
  - Its processing time in M3 is 6
  - Its priority is 1
  - Its due date is 90
- Job3
  - It starts in Queue1 and it has to visit Machine1 and Machine3 (in this sequence) before it exits the system
  - Its processing time in M1 is 10
  - Its processing time in M3 is 3
  - Its priority is 0
  - Its due date is 110

We see above two new attributes of the Job class. These are in reality optional arguments of the parent class (Entity).

- *priority* is an integer. The higher the value the higher the priority assigned to the Entity.
- *dueDate* is a float. It shows the time that the Entity should be out of the system (in case the Entity represents an order or something similar). If our simulation units are minutes and the due date is in exactly one week after the start of the simulation run, then  $dueDate=10080$  ( $60*24*7$ ).

As we see, all the Jobs start from Queue1. The default scheduling rule of a Queue object is FIFO, i.e. the Entity to arrive first in the Queue will be the first to be given in another CoreObject.

Nevertheless, there are several more scheduling rules supported.

- **Priority**: the Entities are sorted in order of ascending predefined priority (the lowest priority is to leave the Queue first)
- **EDD**: the Entities are sorted in order of ascending predefined due date (Earliest Due Date)
- **EOD**: the Entities are sorted in order of ascending predefined order date (Earliest Order Date)
- **NumStages**: the Entities are sorted in order of descending number of stages that they have to pass.
- **RPC**: the Entities are sorted in order of descending total processing time of stages that they have to pass (Remaining Processing Time).
- **MinSlack**: the Entities are sorted in order of ascending slack time. Slack time is defined as due date minus the remaining processing time

- **NextStage:** the Entities are sorted in order of ascending number of Entities in the next stage that the Entity has to pass through.
- **MC:** This stands for Multiple Criteria and it is applied when we have many scheduling rules used. For example we may need to use Priority, but for the Entities that have equal *priorities* EDD will be applied.

(Note: An advanced user can add new scheduling rules by creating a CoreObject that inherits from Queue and overrides the *activeQSorter* method)

We start our model with the assumption that Priority is applied as scheduling rule in Queue1. The other 2 Queues will remain FIFO.

Below is the ManPy main script to run this model (dream\simulation\Examples\JobShop2Priority.py):

```
from SimPy.Simulation import simulate, activate, initialize, infinity, now
from simulation.MachineJobShop import MachineJobShop
from simulation.QueueJobShop import QueueJobShop
from simulation.ExitJobShop import ExitJobShop
from simulation.Job import Job
from simulation.Globals import G

#define the objects of the model
Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="Priority")
Q2=QueueJobShop('Q2','Queue2', capacity=infinity)
Q3=QueueJobShop('Q3','Queue3', capacity=infinity)
M1=MachineJobShop('M1','Machine1')
M2=MachineJobShop('M2','Machine2')
M3=MachineJobShop('M3','Machine3')
E=ExitJobShop('E','Exit')

G.ObjList=[M1,M2,M3,Q1,Q2,Q3,E] #add all the objects in G.ObjList so that they can be
easier accessed later

#define predecessors and successors for the objects
Q1.defineRouting(successorList=[M1])
Q2.defineRouting(successorList=[M2])
Q3.defineRouting(successorList=[M3])
M1.defineRouting(predecessorList=[Q1])
M2.defineRouting(predecessorList=[Q2])
M3.defineRouting(predecessorList=[Q3])

#define the Jobs
J1=Job('J1','Job1',[[ 'Q1',1],[ 'Q3',3],[ 'Q2',2],[ 'E',0]], priority=1, dueDate=100)
J2=Job('J2','Job2',[[ 'Q1',2],[ 'Q2',4],[ 'Q3',6],[ 'E',0]], priority=1, dueDate=90)
J3=Job('J3','Job3',[[ 'Q1',10],[ 'Q3',3],[ 'E',0]], priority=0, dueDate=110)

G.JobList=[J1,J2,J3]

G.maxSimTime=1440.0 #set G.maxSimTime 1440.0 minutes (1 day)

initialize() #initialize the simulation (SimPy method)

#initialize all the objects
for object in G.ObjList:
    object.initialize()

#initialize all the jobs
for job in G.JobList:
```

```

    job.initialize()

#set the WIP for all the jobs
for job in G.JobList:
    Q1.getActiveObjectQueue().append(job)
    job.remainingRoute[0][0]='' #remove data from the remaining route.
    job.schedule.append(['Q1',now()]) #add the data in the schedule that the Job
    entered Q1 at time=0

#activate all the objects
for object in G.ObjList:
    activate(object, object.run())

simulate(until=G.maxSimTime) #run the simulation

#output the schedule of every job
for job in G.JobList:
    #loop in the schedule to print the results
    for record in job.schedule:
        #schedule holds ids of objects. The following loop will identify the name of
        the CoreObject with the given id
        name=None
        for obj in G.ObjList:
            if obj.id==record[0]:
                name=obj.objName
        print job.name, "got into", name, "at", record[1]
    print "-"*30

```

Running the model we get the following in our console:

```

Job1 got into Queue1 at 0
Job1 got into Machine1 at 10
Job1 got into Queue3 at 11
Job1 got into Machine3 at 13
Job1 got into Queue2 at 16
Job1 got into Machine2 at 17
Job1 got into Exit at 19
-----
Job2 got into Queue1 at 0
Job2 got into Machine1 at 11
Job2 got into Queue2 at 13
Job2 got into Machine2 at 13
Job2 got into Queue3 at 17
Job2 got into Machine3 at 17
Job2 got into Exit at 23
-----
Job3 got into Queue1 at 0
Job3 got into Machine1 at 0
Job3 got into Queue3 at 10
Job3 got into Machine3 at 10
Job3 got into Exit at 13
-----

```

We see that Job3 having the highest (lowest value) *priority* was the first to go to Machine1. The other two Jobs had equal priorities, so FIFO was applied (observing the loop where the WIP is set one can see the Job1 was added to Queue1 before Job2).

To test how the model works if Queue1 follows the Earliest Due Date rule we have only to change the definition of Queue1 in our code (dream\simulation\Examples\JobShop2EDD.py):



```
Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="EDD")
```

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 2
Job1 got into Queue3 at 3
Job1 got into Machine3 at 3
Job1 got into Queue2 at 6
Job1 got into Machine2 at 13
Job1 got into Exit at 15
-----
Job2 got into Queue1 at 0
Job2 got into Machine1 at 0
Job2 got into Queue2 at 2
Job2 got into Machine2 at 2
Job2 got into Queue3 at 13
Job2 got into Machine3 at 16
Job2 got into Exit at 22
-----
Job3 got into Queue1 at 0
Job3 got into Machine1 at 3
Job3 got into Queue3 at 13
Job3 got into Machine3 at 13
Job3 got into Exit at 16
-----
```

We see that Job2 having the earliest *dueDate* was the first to go to Machine1. Then Job1 followed and Job3 was the last.

To test how the model works if Queue1 follows the Remaining Process Time rule we have only to change the definition of Queue1 in our code (dream\simulation\Examples\JobShop2RPC.py):

```
Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="RPC")
```

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 12
Job1 got into Queue3 at 13
Job1 got into Machine3 at 13
Job1 got into Queue2 at 16
Job1 got into Machine2 at 16
Job1 got into Exit at 18
-----
Job2 got into Queue1 at 0
Job2 got into Machine1 at 10
Job2 got into Queue2 at 12
Job2 got into Machine2 at 12
Job2 got into Queue3 at 16
Job2 got into Machine3 at 16
Job2 got into Exit at 22
-----
Job3 got into Queue1 at 0
Job3 got into Machine1 at 0
Job3 got into Queue3 at 10
Job3 got into Machine3 at 10
Job3 got into Exit at 13
-----
```

We see that Job3 having the greatest remaining processing time was the first to go to Machine1. Then Job2 followed and Job1 was the last.

Finally, we want to test how the model works if Queue1 follows a multi criteria rule. First Priority is applied, and if Jobs have equal priorities, then EDD is applied we have only to change the definition of Queue1 in our code (dream\simulation\Examples\JobShop2MC.py):

```
Q1=QueueJobShop('Q1','Queue1', capacity=infinity, schedulingRule="MC-Priority-EDD")
```

We see that to define a multi criteria rule, we use MC and then the scheduling rules according to their sequence. All the scheduling rules are separated with “-”.

Running the model we get the following in our console:

```
Job1 got into Queue1 at 0
Job1 got into Machine1 at 12
Job1 got into Queue3 at 13
Job1 got into Machine3 at 13
Job1 got into Queue2 at 16
Job1 got into Machine2 at 16
Job1 got into Exit at 18
-----
Job2 got into Queue1 at 0
Job2 got into Machine1 at 10
Job2 got into Queue2 at 12
Job2 got into Machine2 at 12
Job2 got into Queue3 at 16
Job2 got into Machine3 at 16
Job2 got into Exit at 22
-----
Job3 got into Queue1 at 0
Job3 got into Machine1 at 0
Job3 got into Queue3 at 10
Job3 got into Machine3 at 10
Job3 got into Exit at 13
-----
```

We see that having the highest (lowest value) *priority*, Job3 was the first to get into Machine1. Contrary to the first example of this subsection though (JobShop2Priority.py), now Job2 is the second Job to go to the Machine. This happens because it has an earlier due date (*dueDate*) than Job1.

## 4.9 Output trace to Excel

Simulation programs are notoriously difficult to debug. One asset that ManPy objects offer in order to enhance debugging is the feature of outputting trace to Excel. All ManPy objects output to the same Excel file and the events are sorted in increasing timestamp. The trace is essential for debugging. To run a model that is believed to be verified, it should be turned off since it slows the program significantly.

`dream\simulation\Examples\JobShop1Trace.py` runs the same model as `dream\simulation\Examples\JobShop1.py` with the difference that trace is outputted. In order to achieve this 3 additions are required:

- Import `ExcelHandler` (script that holds Excel related methods.):

```
import simulation.ExcelHandler
```

- Set the global `trace` variable to 'Yes' (note that this is case sensitive) at the start of the main script:

```
G.trace="Yes"
```

- Call the `outputTrace` method of the `ExcelHandler` at the end of the main script. The argument is optional and gives the filename, (default value is 'Trace')

```
simulation.ExcelHandler.outputTrace('TRACE')
```

Running the script `TRACE.xls` will be created with the following contents:

0	Job1	releases Queue1
0	Job1	got into Machine1
1	Job1	ended processing in Machine1
1	Job1	releases Machine1
1	Job1	got into Queue3
1	Job1	releases Queue3
1	Job1	got into Machine3
4	Job1	ended processing in Machine3
4	Job1	releases Machine3
4	Job1	got into Queue2
4	Job1	releases Queue2
4	Job1	got into Machine2
6	Job1	ended processing in Machine2
6	Job1	releases Machine2
6	Job1	exited the system

Note that every object has its own `outputTrace` method which a user can customize. Of course this can also be omitted if it is not desirable for the object to output trace at all.